Politechnika Wrocławska

Sztuczna inteligencja i inżynieria wiedzy

Laboratorium - Lista 1

21.03.2024

Krzysztof Głowacz, 266545

Spis treści

1 Przygotowanie danych

Uwaga techniczna: całość implementacji została wykonana w języku ***Python****.*

W celu wydajnego przetwarzania danych w dalszej części laboratorium na początku przeprowadzono *preprocessing*, który polegał na zbudowaniu grafu, gdzie wierzchołkiem był przystanek o atrybutach: nazwa, długość i szerokość geograficzna, a krawędziami były połączenia komunikacyjne, jakie można wykonać między dwoma wierzchołkami, przy czym każde połączenie było opisane za pomocą: symbolu linii, czasu odjazdu i czasu przyjazdu. Implementacja została oparta na strukturze danych o stałoczasowym dostępie do elementów, aby zapewnić jak najmniejszą złożoność czasową operacji, która była w największym stopniu wykorzystywana przez algorytmy przeszukujące. Dodatkowo w czasie przetwarzania wszystkich połączeń z pliku obliczana jest sumaryczna odległość pokonana przez wszystkie linie oraz sumaryczny czas. Następnie obliczany jest iloraz czasu i odległości, który wykorzystywany jest przy algorytmie $A*$. Zazwyczaj pełne przetworzenie pliku, tzn. pobranie każdej linii, dokonanie konwersji do odpowiednich typów (przede wszystkim daty i godziny do obiektu typu *datetime*) i dołożenie jej do budowanego grafu zajmuje od 20 do 50 sekund, w zależności od mocy użytego urządzenia.

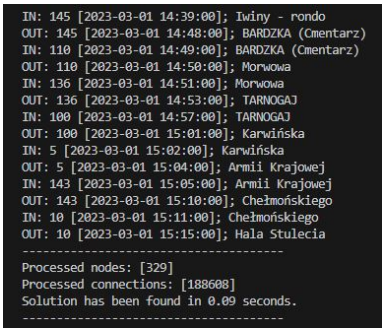
2 Algorytm Dijkstry

Pierwszym elementem laboratorium było zaimplementowanie algorytmu Dijkstry w celu znalezienia najkrótszej trasy z przystanku początkowego do przystanku końcowego w oparciu o kryterium czasu. W celach testowych zaimplementowano także algorytm dla kryterium przesiadek. Przy implementacji samego algorytmu posłużyłem się pseudokodem zamieszczonym na stronie Wikipedii [1]. Kluczowe punkty tego algorytmu to:

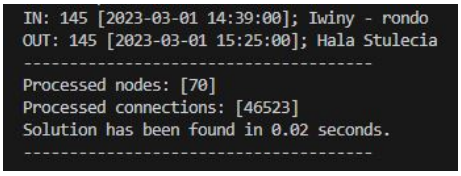
- Wybór wierzchołka o najmniejszej wadze ze zbioru wierzchołków jeszcze nieprzetworzonych. Wagą wierzchołka jest odpowiednio najszybszy możliwy czas pojawienia się na przystanku (jednak późniejszy niż czas rozpoczęcia podróży) lub liczba dokonanych przesiadek licząc od przystanku początkowego podróży.
- Wyszukanie najlepszego połączenia między dwoma przystankami, zależnie od kryterium. Dla kryterium czasu będzie to najszybsze połączenie, które odjeżdża najwcześniej o godzinie pojawienia się na przystanku w przypadku braku zmiany linii lub w przypadku zmiany linii - o godzinie pojawienia się na przystanku powiększonej o zdefiniowaną stałą oznaczającą minimalny wymagany czas na przesiadkę. Dla kryterium przesiadek będzie to najszybsze połączenie niewymagające przesiadki, które odjeżdża najwcześniej o godzinie pojawienia się na przystanku lub, w przypadku braku wspomnianego połączenia, najlepsze połączenie zdefiniowane dla kryterium czasu.

Algorytm działa poprawnie dla większości przypadków. Poniższe zrzuty ekranu obrazują przykładowe wyniki na trasie:

$$1.03.2024,\ Iwiny - rondo \rightarrow Hala\ Stulecia,\ godz.\ 14:38$$



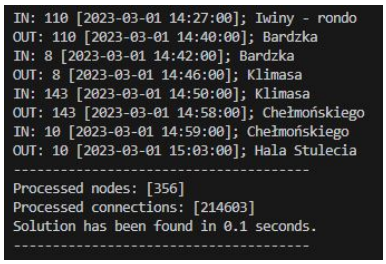
(a) Rezultat dla kryterium czasu



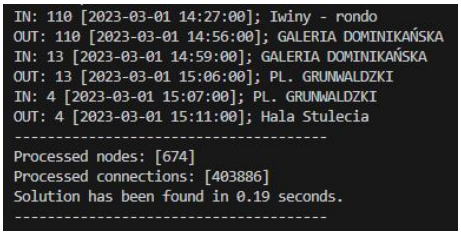
(b) Rezultat dla kryterium przesiadek

Zdjęcie 1: Wyniki algorytmu Dijkstry zgodne z oczekiwaniami

Rezultaty te są zgodne z oczekiwaniami w obu przypadkach. Należy jednak pamiętać, że ze względu na zachłanność algorytmu Dijkstry trudne jest poprawne zaimplementowanie dobrego algorytmu do kryterium przesiadek. Niech za przykład obrazujący problem posłużą poniższe zdjęcia:



(a) Rezultat dla kryterium czasu



(b) Zły rezultat dla kryterium przesiadek

Zdjęcie 2: Wyniki algorytmu Dijkstry - nieoptymalne kryterium przesiadek

Powyższe rezultaty pochodzą z eksperymentu, w którym zmieniona została poprzednia godzina odjazdu o 13 minut (z godz. 14:38 na 14:25). Dla kryterium czasu został znaleziony wynik o takim samym czasie podróży, jak w poprzedniej sytuacji. Dla kryterium przesiadek natomiast wynik przestał być optymalny - zamiast bezpośredniego połączenia linią 145 zaproponowana została podróż z dwiema przesiadkami. Wynika to z faktu, że algorytm znajdując się na danym przystanku nie wie, które dokładnie linie prowadzą bezpośrednio do przystanku końcowego. Jeśli więc z danego przystanku odjeżdżają przykładowo dwie linie w tym samym kierunku, a skorzystanie z obu wymaga przesiadki, to algorytm wybierze to połączenie, które odbywa się wcześniej bez względu na to, że być może to drugie byłoby bezpośrednim połączeniem z przystankiem docelowym.

3 Algorytm A*

Głównym elementem laboratorium była implementacja algorytmu A*, a więc de facto zaproponowanie odpowiedniej heurystyki, dzięki której algorytm byłby wydajniejszy od algorytmu Dijkstry. Idee stojące za użytymi heurystykami opisane są w poniższych sekcjach.

3.1 Heurystyka - kryterium czasowe

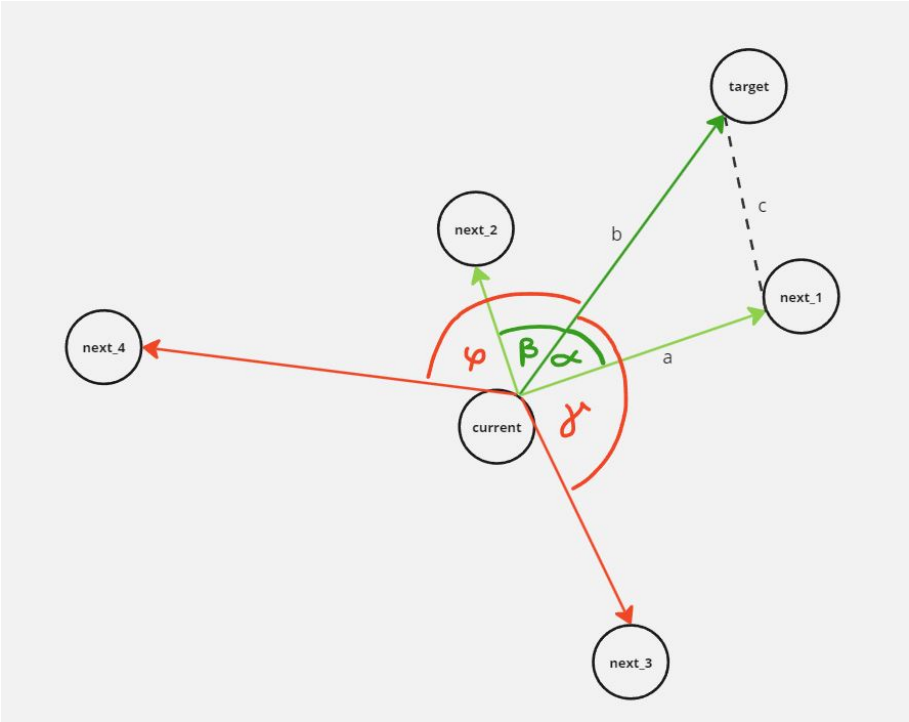
Znalezienie heurystyki dla kryterium czasowego było stosunkowo prostym zadaniem. W tym przypadku heurystyka miała być funkcją, która pozwoli optymistycznie aproksymować odległość rozważanego sąsiedniego przystanku od przystanku docelowego podróży. Wykorzystując relaksację problemu obliczana jest w tym przypadku odległość euklidesowa między przystankami na podstawie ich współrzędnych geograficznych. Następnie wynik mnożony jest przez średni czas potrzebny na pokonanie stopnia geograficznego. Otrzymany czas nie będzie jednak gwarantował, że heurystyka jest akceptowalna - może się okazać, że akurat na danej trasie średni czas pokonania stopnia geograficznego jest mniejszy niż średnia globalna, więc wynik byłby przeszacowany. Aby wyeliminować taką sytuację całość mnożona jest jeszcze przez eksperymentalnie wyznaczony mnożnik (równy 0.5), który gwarantuje, że heurystyka faktycznie będzie spełniać podstawowe założenia. Dodatkowo na początku sprawdzane jest, czy przetwarzany sąsiedni przystanek nie jest przystankiem docelowym. Jeśli tak, to automatycznie heurystyka zwraca wartość 0. Funkcja ta wygląda następująco:

```
def heuristic_time(graph: dict[str, Node], node: str, target: str, average_speed: float):
    if node == target:
        return timedelta(minutes=0)
    start_lat = graph[node].latitude
    start_lon = graph[node].longitude
    end_lat = graph[target].latitude
    end_lon = graph[target].longitude
    dist = sqrt((end_lat - start_lat) ** 2 + (end_lon - start_lon) ** 2)
    return timedelta(minutes=(dist * average_speed * config.TIME_HEURISTIC_AVG_FACTOR))
```

Dzięki początkowemu sprawdzeniu, że nie mamy jeszcze do czynienia z przystankiem końcowym mamy pewność, że istotnie nastąpi relaksacja problemu, ponieważ rozważać będziemy połączenia wymagające odwiedzenia po drodze do celu jeszcze przynajmniej jednego przystanku pośredniego. Tak zaprojektowana funkcja faktycznie spełnia kluczowe założenia heurystyki.

3.2 Heurystyka - kryterium przesiadkowe

Dla kryterium przesiadkowego znalezienie heurystyki było zdecydowanie trudniejsze. Projektując odpowiednią funkcję należało to zrobić tak, by aproksymowała ona liczbę przesiadek koniecznych do wykonania od rozważanego przystanku sąsiedniego do przystanku końcowego. Jedynym pomysłem, jaki spełniał to założenie, był ten wykorzystujący powstające kąty. Aby zobrazować to rozwiązanie wykorzystany zostanie poglądowy diagram:



Zdjęcie 3: Poglądowy diagram - wstęp do kryterium przesiadek

Podstawą aproksymowania liczby przesiadek jest w tym przypadku kąt powstający między wektorem prowadzącym z aktualnego przystanku do rozważanego sąsiada, a wektorem prowadzącym bezpośrednio od aktualnego przystanku do przystanku docelowego. Skoro współrzędne każdego z przystanków są jawne, tzn. znamy wszystkie długości oznaczone jako a, b, c, to możemy skorzystać z twierdzenia cosinusów ($cos\alpha = \frac{c^2-a^2-b^2}{-2ab}$), aby poznać wartość funkcji cosinus dla powstałego kąta. Skoro jednak nie wiemy, czy przystanek docelowy leży na trasie akurat wybranego kursu, to nie możemy zwracać dodatniej wartości funkcji heurystyki dla sąsiadów aktualnego wierzchołka, do których przemieszczamy się tym samym kursem. Łatwo wyobrazić sobie kontrprzykład, który obrazowałby fakt powstania niechcianego (tj. rozwartego) kąta między przystankami wchodzącymi w skład tej samej linii, przy czym dojechanie do przystanku docelowego nie wymaga wykonywania przesiadki (premiujemy w ten sposób połączenia w ramach tego samego kursu). Jeśli jednak połączenie prowadzące z aktualnego przystanku do przystanku docelowego wymaga wykonania przesiadki, to możemy odpowiednio ocenić słuszność tego posunięcia. Jeśli bowiem znalezione połączenie opiera się na przesiadce, a następnie przejeździe w „drugą stronę" (gdy powstały kąt jest przykładowo rozwarty), to prawdopodobnie poza właśnie dokonaną przesiadką konieczne będzie wykonanie jeszcze jednej umożliwiającej powrót na właściwy kurs. Implementacja opiera się na sprawdzeniu, czy w danym kroku dokonywana jest przesiadka oraz czy cosinus powstałego kąta jest mniejszy niż $\frac{1}{2}$, tzn. czy powstały kąt jest większy niż $60°$. Jeśli tak, to dokonywana jest normalizacja wartości cosinusa, by ta była liczbą z zakresu $[0, 1\frac{1}{2})$, po czym liczba ta jest odejmowana od maksymalnej liczby tzn. od $1\frac{1}{2}$ i zwracana jako wynik heurystyki. W przeciwnym przypadku zwracana wartość to 0. Implementacja wygląda następująco:

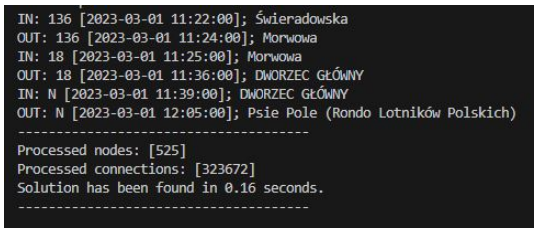
```python
def cosinus_between_two_vectors(graph: dict[str, Node], current: str, _next: str, target: str):
    x_current, y_current = graph[current].latitude, graph[current].longitude
    x_next, y_next = graph[_next].latitude, graph[_next].longitude
    x_target, y_target = graph[target].latitude, graph[target].longitude
    current_next = sqrt((x_next - x_current) ** 2 + (y_next - y_current) ** 2)
    current_target = sqrt((x_target - x_current) ** 2 + (y_target - y_current) ** 2)
    next_target = sqrt((x_target - x_next) ** 2 + (y_target - y_next) ** 2)
    return ((next_target ** 2 - current_next ** 2 - current_target ** 2)) / (-2.0 *
        current_next * current_target)

def heuristic_line_change(graph, was_change: bool, current: str, _next: str, target: str):
    cosinus = Astar.cosinus_between_two_vectors(graph, current, _next, target)
    if was_change and cosinus < 0.5:
        scaled_cos = cosinus + 1.0
        return (1.5 - scaled_cos)
    return 0
```

3.3 Porównanie z algorytmem Dijkstry

Po zaimplementowaniu przedstawionych heurystyk, a także zmodyfikowaniu algorytmu, by jak najwcześniej wykrywał zbliżenie się do przystanku docelowego na odległość bezpośredniego połączenia, można śmiało stwierdzić, że algorytm A* w takiej formie jest obiektywnie lepszy i wydajniejszy niż algorytm Dijkstry. Do porównania posłużą nam wyniki 4 eksperymentów przeprowadzonych na trasie:

1.03.2024, *Świeradowska* → *Psie Pole* (*Rondo Lotników Polskich*), *godz.* 11 : 15

(a) Dijkstra - kryterium czasu



(b) Dijkstra - kryterium przesiadek

(c) A* - kryterium czasu

(d) A* - kryterium przesiadek

Zdjęcie 4: Porównanie wyników obu algorytmów

Widzimy więc, że algorytm A* w obu przypadkach zwrócił to samo rozwiązanie, co algorytm Dijkstry, ale w obu przypadkach zrobił to szybciej, mniejszym kosztem. Do znalezienia rozwiązania potrzebował odwiedzić ok. 40-60% węzłów (przystanków) mniej i przetworzyć podobnie mniejszą liczbę połączeń. Niestety niektóre eksperymenty wykazały, że algorytm A* zwracał gorsze wyniki w przypadku kryterium przesiadek niż algorytm Dijkstry wykonując przykładowo jedną przesiadkę więcej. Były to mimo wszystko nieliczne przypadki wśród wielu rezultatów zgodnych z oczekiwaniami. Prawdopodobnie przygotowana heurystyka nie jest w 100% akceptowalna i zdarzają się sytuacje, w których algorytm nie

osiągnie rozwiązania optymalnego przez pesymistyczną aproksymację heurystyki. Natomiast w każdym przeprowadzonym eksperymencie widoczne było wyraźnie szybsze wykonanie algorytmu A* niezależnie od przyjętego kryterium.

4 Tabu search

Ostatnim wykonanym zadaniem było to związane z algorytmem Tabu search. Program napisany w tym zadaniu miał na celu znalezienie najkrótszej trasy rozpoczynającej się w zadanym przystanku, przebiegającej przez wszystkie przystanki z podanej listy i wracającej do przystanku początkowego. Podobnie jak w poprzednich zadaniach należało wziąć pod uwagę dwa kryteria: czasu i przesiadek. Mając zaimplementowany wydajny algorytm wyszukujący najkrótszą trasę między dwoma przystankami (algorytm A*) problem ten sprowadzał się do znalezienia odpowiedniej permutacji ciągu przystanków, dla której koszt całej trasy (tzn. czas przyjazdu na przystanek początkowy ostatnią trasą w kryterium czasu lub suma przesiadek w kryterium przesiadek) jest najmniejszy. Implementacja tego zadania oparta została na pseudokodzie dołączonym do instrukcji laboratorium.

Początkowy krok, tzn. wygenerowanie pierwszego rozwiązania polegał na posortowaniu przystanków według ich odległości euklidesowej od przystanku początkowego, a następnie ustawieniu ich w ciąg w takiej kolejności, aby przystanek najbardziej oddalony znalazł się w środku, a następnie idąc od niego w lewo i prawo w stronę przystanku początkowego znajdowały się przystanki coraz bliższe przystankowi początkowego (w celu stworzenia czegoś na kształt pętli). Następnie do osiągnięcia warunku końcowego, tzn. do przekroczenia limitu wykonania pętli określonego w pliku konfiguracyjnym, następowało generowanie sąsiednich rozwiązań, tj. rozwiązań, w których zamieniane miejscami były przystanki leżące na trasie z obu stron między przystankiem początkowym, a przystankiem będącym w środku listy (na wypadek gdyby stworzona pętla była „zagmatwana" i wymagała „rozplątania"). Dalej następowała ewaluacja sąsiadów i w razie konieczności odpowiednie zamiany w celu zapamiętania ostatecznie najlepszego rozwiązania. Do ewaluacji używany był wspomniany algorytm A*. Na końcu danej iteracji sprawdzone rozwiązania trafiały do zbioru rozwiązań zakazanych (tabu) w celu uniknięcia ich ponownego sprawdzania. Co jakiś czas następowało też przetasowanie polegające na przesunięciu cyklicznym wszystkich przystanków tworzących pętlę (poza przystankiem początkowym) tak, aby algorytm próbował wielu kombinacji uporządkowania przystanków (i wielu kształtów tworzącej się pętli). Wyniki przykładowego eksperymentu przedstawione są na kolejnej stronie. Dane w obu eksperymentach były następujące:

A = Iwiny - rondo
L = [Hynka, KRZYKI, Jastrzębia, Bacciarellego, Swojczyce, Rynek]

Zwrócone wyniki różnią się od siebie, dla kryterium czasu wynikiem był ciąg:

s* = [Iwiny - rondo, Hynka, Jastrzębia, KRZYKI, Bacciarellego, Swojczyce, Rynek, Iwiny - rondo]

Dla kryterium przesiadek natomiast:

s* = [Iwiny - rondo, Rynek, Swojczyce, Bacciarellego, KRZYKI, Jastrzębia, Hynka, Iwiny - rondo]

Szczegóły każdej trasy, wykonane przesiadki, czasy przejazdów widoczne są na zdjęciu 5.

Literatura

[1] Wikipedia, Dijkstra's algorithm
`https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`. Dostęp: 19.03.2024.

```
Path from Iwiny - rondo to Hynka:
IN: 110 [2023-03-01 08:45:00]; Iwiny - rondo
OUT: 110 [2023-03-01 08:56:00]; Krynicka
IN: 21 [2023-03-01 08:57:00]; Krynicka
OUT: 21 [2023-03-01 09:25:00]; Kwiska
IN: 128  [2023-03-01 09:26:00]; Kwiska
OUT: 128  [2023-03-01 09:32:00]; Hynka

Path from Hynka to Jastrzębia:
IN: 136 [2023-03-01 09:34:00]; Hynka
OUT: 136 [2023-03-01 09:56:00]; Hallera
IN: 14 [2023-03-01 09:57:00]; Hallera
OUT: 14 [2023-03-01 09:58:00]; Jastrzębia

Path from Jastrzębia to KRZYKI:
IN: 14 [2023-03-01 09:58:00]; Jastrzębia
OUT: 14 [2023-03-01 10:02:00]; KRZYKI

Path from KRZYKI to Bacciarellego:
IN: 14 [2023-03-01 10:02:00]; KRZYKI
OUT: 14 [2023-03-01 10:06:00]; Hallera
IN: 143 [2023-03-01 10:07:00]; Hallera
OUT: 143 [2023-03-01 10:32:00]; Chełmońskiego
IN: 145 [2023-03-01 10:35:00]; Chełmońskiego
OUT: 145 [2023-03-01 10:38:00]; Monte Cassino
IN: 146 [2023-03-01 10:39:00]; Monte Cassino
OUT: 146 [2023-03-01 10:40:00]; Bacciarellego
```

(a) Tabu search - kryterium czasu, cz.1

```
Path from Bacciarellego to Swojczyce:
IN: 146 [2023-03-01 10:40:00]; Bacciarellego
OUT: 146 [2023-03-01 10:42:00]; BARTOSZOWICE
IN: 345 [2023-03-01 10:45:00]; BARTOSZOWICE
OUT: 345 [2023-03-01 10:49:00]; Chełmońskiego
IN: 145 [2023-03-01 10:50:00]; Chełmońskiego
OUT: 145 [2023-03-01 10:56:00]; SĘPOLNO
IN: 115 [2023-03-01 10:58:00]; SĘPOLNO
OUT: 115 [2023-03-01 11:03:00]; Swojczyce

Path from Swojczyce to Rynek:
IN: 115 [2023-03-01 11:03:00]; Swojczyce
OUT: 115 [2023-03-01 11:16:00]; PL. GRUNWALDZKI
IN: D [2023-03-01 11:17:00]; PL. GRUNWALDZKI
OUT: D [2023-03-01 11:24:00]; GALERIA DOMINIKAŃSKA
IN: 13 [2023-03-01 11:25:00]; GALERIA DOMINIKAŃSKA
OUT: 13 [2023-03-01 11:27:00]; Świdnicka
IN: K [2023-03-01 11:29:00]; Świdnicka
OUT: K [2023-03-01 11:31:00]; Rynek

Path from Rynek to Iwiny - rondo:
IN: 7 [2023-03-01 11:33:00]; Rynek
OUT: 7 [2023-03-01 11:40:00]; Arkady (Capitol)
IN: 21 [2023-03-01 11:41:00]; Arkady (Capitol)
OUT: 21 [2023-03-01 11:54:00]; Morwowa
IN: 145 [2023-03-01 11:57:00]; Morwowa
OUT: 145 [2023-03-01 12:07:00]; Iwiny - rondo
```

(b) Tabu search - kryterium czasu, cz.2

```
Path from Iwiny - rondo to Rynek:
IN: 110 [2023-03-01 08:45:00]; Iwiny - rondo
OUT: 110 [2023-03-01 09:13:00]; GALERIA DOMINIKAŃSKA
IN: 10 [2023-03-01 09:16:00]; GALERIA DOMINIKAŃSKA
OUT: 10 [2023-03-01 09:18:00]; Świdnicka
IN: K [2023-03-01 09:22:00]; Świdnicka
OUT: K [2023-03-01 09:24:00]; Rynek

Path from Rynek to Swojczyce:
IN: K [2023-03-01 09:25:00]; Rynek
OUT: K [2023-03-01 09:30:00]; GALERIA DOMINIKAŃSKA
IN: N [2023-03-01 09:33:00]; GALERIA DOMINIKAŃSKA
OUT: N [2023-03-01 09:51:00]; Brücknera
IN: 118 [2023-03-01 09:53:00]; Brücknera
OUT: 118 [2023-03-01 10:04:00]; Swojczyce

Path from Swojczyce to Bacciarellego:
IN: 118 [2023-03-01 10:04:00]; Swojczyce
OUT: 118 [2023-03-01 10:09:00]; SĘPOLNO
IN: 145 [2023-03-01 10:15:00]; SĘPOLNO
OUT: 145 [2023-03-01 10:17:00]; Libelta
IN: 345 [2023-03-01 10:41:00]; Libelta
OUT: 345 [2023-03-01 10:43:00]; Bacciarellego

Path from Bacciarellego to KRZYKI:
IN: 345 [2023-03-01 10:43:00]; Bacciarellego
OUT: 345 [2023-03-01 10:56:00]; Kochanowskiego
IN: D [2023-03-01 10:58:00]; Kochanowskiego
OUT: D [2023-03-01 11:22:00]; Hallera
IN: 602 [2023-03-01 13:46:00]; Hallera
OUT: 602 [2023-03-01 13:49:00]; KRZYKI
```

(c) Tabu search - kryterium przesiadek, cz.1

```
Path from KRZYKI to Jastrzębia:
IN: 602 [2023-03-01 15:10:00]; KRZYKI
OUT: 602 [2023-03-01 15:13:00]; Hallera
IN: 14 [2023-03-01 15:15:00]; Hallera
OUT: 14 [2023-03-01 15:16:00]; Jastrzębia

Path from Jastrzębia to Hynka:
IN: 6 [2023-03-01 15:18:00]; Jastrzębia
OUT: 6 [2023-03-01 15:21:00]; Rondo
IN: 126 [2023-03-01 15:23:00]; Rondo
OUT: 126 [2023-03-01 15:59:00]; Bystrzycka
IN: 136 [2023-03-01 16:00:00]; Bystrzycka
OUT: 136 [2023-03-01 16:01:00]; Hynka

Path from Hynka to Iwiny - rondo:
IN: 136 [2023-03-01 16:01:00]; Hynka
OUT: 136 [2023-03-01 16:32:00]; Morwowa
IN: 110 [2023-03-01 16:40:00]; Morwowa
OUT: 110 [2023-03-01 16:55:00]; Iwiny - rondo
```

(d) Tabu search - kryterium przesiadek, cz.2

Zdjęcie 5: Porównanie wyników Tabu search dla obu kryteriów